INF 4130: Execises to Matchings and Flow

October 31, 2013

With answers

Note: The words "vertex" and "node" are used with the same meaning in this document.

Exercise 1

Solve Exercise 14.4 in the text book (B&P) (and sketch a data structure for Exercise 14.5).

The exercise is to show that the Hungarian Algorithm can be implemented in time $O(n^3)$ for a bipartite graph $G = (X, Y, E)$, with $|X| = |Y| = n$.

If we think about the algorithm, without studying the code on pages 422-423, it consists of an outer loop where we repeatedly find and apply augmenting paths. Applying an augmenting path increases the size our matching with one edge (two vertices, one from X and one from Y). We can therefore at most iterate through this outer loop n times.
Inside the outer loop we build a tree, edge by edge, in our search for an augmenting path; or rather, we build the tree two and two edges at the time (one matched and one unmatched), unless we find an augmenting path. The order in which we add edges to the tree is arbitrary, what is important is that we can add a new pair of edges in time $O(1)$. We can do that if we 1) have a flag in each vertex that says whether or not the vertex is part of the tree, and 2) have a pointer in each vertex that points to the vertex with which it is matched (or null if it is unmatched). We also need some way to keep vertices in a set, where insertion and removal can be done in time $O(1)$, a linked list is suitable. This set, let us call it R, is used to represent the front of the tree we grow – vertices that are part of the tree, that we not yet have followed edges out from. This set R initially contains only the unmatched vertex r in X we choose to start from – the root. The step is to take a vertex out of R, and follow all edges out from that vertex, each such edge can either:

- Go to a vertex already in the tree. We do nothing.
- Go to a matched vertex outside the tree. We expand the tree with this vertex *and* the vertex with which it is matched, and insert the latter one in R (not the middle vertex).
- Go to an unmatched vertex outside the tree. We have an augmenting path and our tree-building stops.

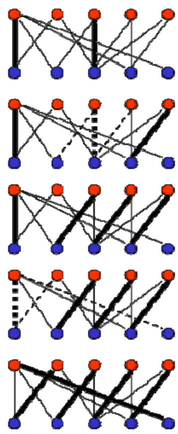
These operations can now be done in time $O(1)$. Since there are no more than n^2 edges, finding an augmenting path (i.e. building the tree) only takes time $O(n^2)$ (each time). All of this gives us a total running time for the algorithm of $O(n^3)$.

```
PROC Hungarian (G = (X,Y,E))   // |E| can be up to n²
{
  M = Ø; // Den tome mengden
  WHILE <not finished> // not perfect matching, possibly new augmenting paths
  {
     P = <new augmenting path>;    // Or terminate if none is found
     M = M ⊕ P; // |M| = |M| + 1
  }
}
```

```
CLASS Vertex
{
   Boolean isInTree = FALSE;
   Vertex matchedWith = null;
   Boolean proc isMatched { matchedWith != null; }
   Vertex previous, next; // for the linked list (R)
   Vertex parentInTree
   …
}
```

Exercise 2

Solve Exercise 14.6 in the textbook.



We start with the graph given in the exercise, at the top of the figure to the right. We number the vertices from left to right x_1, x_2, …, x_5 and y_1, y_2, …, y_5. We start by growing a tree from x_5, and immediately get an augmenting path (of *one* edge) if we look at the edge (x_5, y_4). Remember that an edge with unmatched vertices at both ends is a (simplest possible) augmenting path.

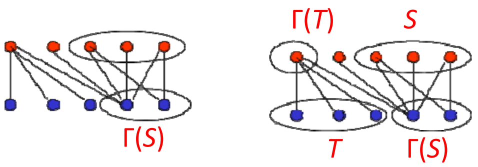
After this augmentation has been done, we can start building a tree from for instance x_4, and one possibility is then that we find the augmenting path x_4-y_3-x_3-y_2 (dotted lines in graph number two).

Applying this augmenting path we get the third graph, and if we then build a tree from x2, we sooner or later find the augmenting path indicated by the dotted lines in graph number four. Applying *that* augmenting path results in the perfect matching of graph number five.

Exercise 3

Assume $|X| = |Y|$. Then show that if we have found a subset S of X with $|\Gamma(S)| < |S|$, we can also easily find a subset of T of Y with $|\Gamma(T)| < |T|$.

This is actually easy to show. Assume we a subset S of X such that $\Gamma(S)$ has fewer vertices than S, as shown in the figure below. By the definition of $\Gamma(S)$ no edge can go between S and $T = Y - \Gamma(S)$. Therefore $\Gamma(T)$ must be a subset (not necessarily proper) of $X - S$, and thereby be smaller than T.



Exercise 4

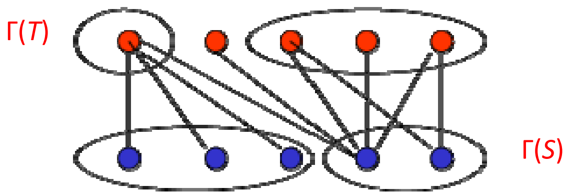
Question 4.a

Show that, for general graphs, any "node cover" (a subset of the nodes that "covers" all the edges) will never have fewer nodes than there are edges in a matching.

Assume that a graph G has a matching M, and a node cover NC. Then each edge in M must have at least one of its end nodes in NC (otherwise NC did not cover all edges). The end nodes of an edge in M must be separate from the end nodes of any other edge in M. Thus the number of nodes in NC must be at least as large as the number of edges i

Question 4.b

Look at some examples with bipartite graphs, and observe that in such graphs you can always find a matching and a node cover of the same size. (It is in fact not difficult to prove this by looking at the situation when the Hungarian stops after having built alternating trees from all unmatched node in X, and no augumenting path is found. The above fact can be used to prove that a certain match is as large as possible, also for cases with $|X| \neq |Y|$).

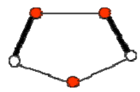


As an example, we can look at the graph from Exercise 3. A cover could be x_1, y_4 and y_5, which is the union of Γ(*T*) and Γ(*S*). This also indicates how a node cover can be found when the matching algorithm stops. A matching with three edges is easy to find, and we then know that this has as many edges as possible.

Question 4.c

Find an example showing that, in *general* graphs, one cannot always find a node cover and a matching of the same size.

Finding a graph where the maximum matching and the minimum vertex cover are of different size is easy. The canonical example of a non-bipartite graph, the odd loop, does the trick. In a C_5, for instance, the largest matching has two edges, while the smallest vertex cover has three nodes.

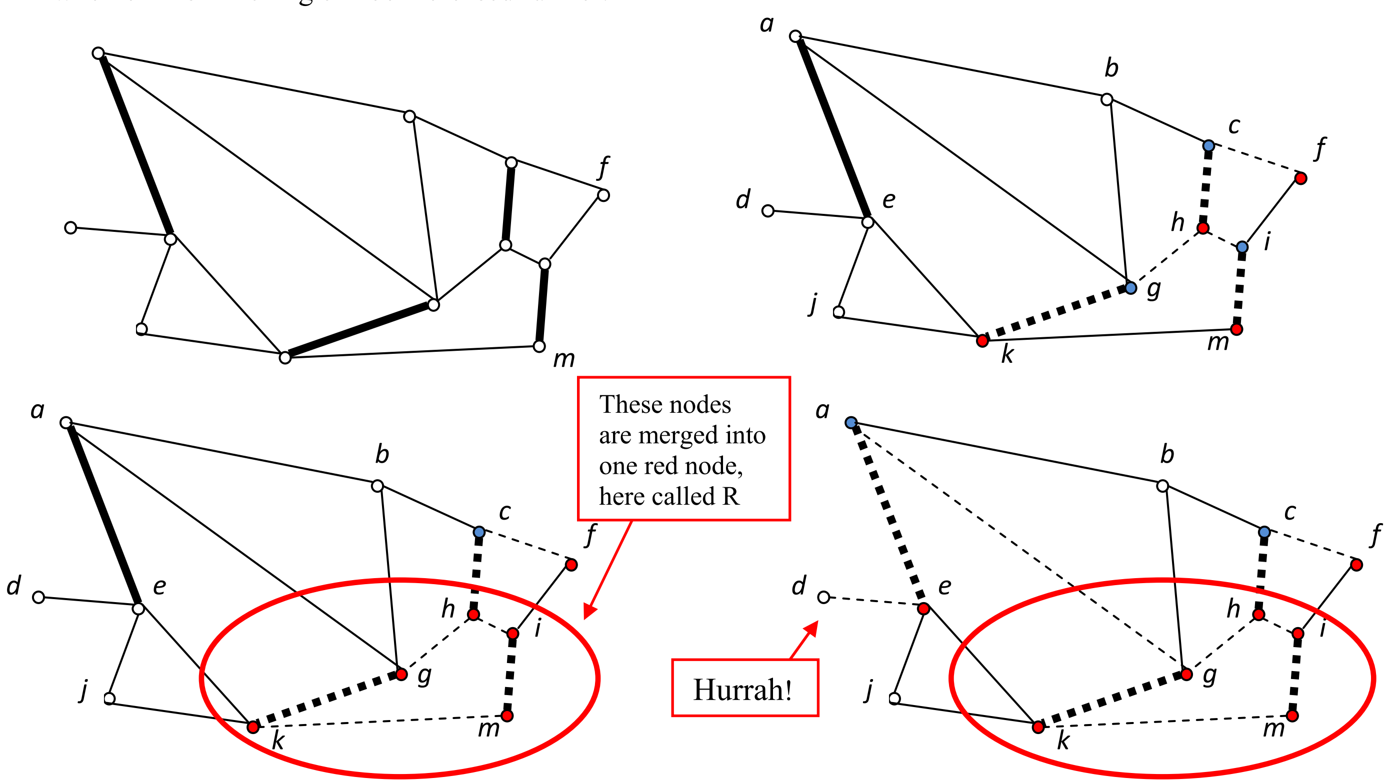


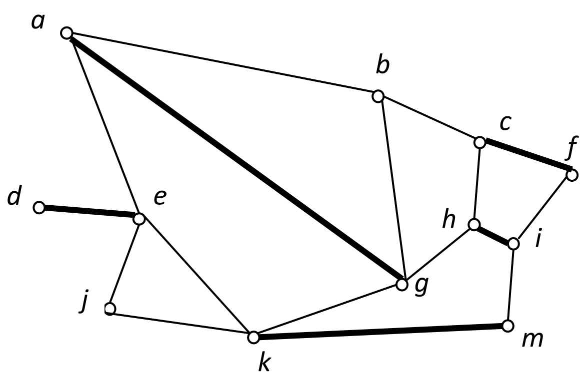
Exercise 5

We are given the following graph G, and given matching M. You shall use the maximum matching algorithm for general graphs to find a maximum matching for G, by starting with M. Start at node *f* as the root, then look at the edge *f*-*c* getting also *c*-*h* into the tree. Then look at edges *h*-*g* and *h*-*i*, which will both increase the tree by two nodes each.

Which nodes are now red and blue (assuming that the root *f* is red)?

Then look at the unmatched edge out of *m*. What will happen then? Proceed with choices so that you end up finding an augumenting path between *d* and *f* (even though one between *b* and *f* or *j* and *f* is closer by). Show the resulting matching after you have "used" this augumenting path. Finally, decide whether this matching can be increased further.





When "using" the found augumenting path (above), the large red node R (made from merging the nodes *h*, *g*, *k*, *i* and *m* in an earlier step), must now be unwrapped, so that we can see that the correct alternating path through it (from the *d* end) is:

g, *k*, *m*, *i*, *h*.

The figure to the left shows the result of "using" the path.

Thus, the size of the matching is increased by one. To see whether it can be increased further, we must remove all node colors, and repeat the tree-building process from all unmatched nodes (that is, from *j* and from *b*). Starting from either, we would find the augumenting path: *j k m i h g a b*, and by using this path we get a perfect matching.

How one would find this augumenting path and how many mergings of odd cycles we would have depends to a large extent on where we start (*j* or *b*) and in what order we look at edges out of red nodes. There is obviously a "lucky" choice of root node and order that would give no mergings at all, by "following" the augumenting path. That is, we e.g. use *b* as the root, and first look at *b*-*a*. This would add *b*-*a* and *a*-*g* to the tree, then look at *g*-*h* that would add *g*-*h* and *h*-*i* to the tree, and so on along the path. On the other hand, if we started at *b* and first looked at *b*-*g* and then at *b*-*a* we would get a merging of *b, a, g* already at that point, and this node would in fact also be the root-node until we unpack it after the augumanting path is found. We might also get a merging of *g* (or rather *gba*), *h, i, m, k* before we finally find the augmenting path.

Exercise 6

To study the max flow algorithm, go through the example in Figure 14.9 in detail (B&P). See introduction at the bottom of page 439. (Note that there are many typos in early editions of the book, most should now be corrected.)

Left to the class. Look at the two last pages of this document.

Known typos in early editions of the textbook are:

- Step 1: Edge 4-7 in N_f should be dotted.
- Step 2—7: Edge 4-7 shoud be reversed in all N_fs.
- Step 2: Inner edges in the flow graph should be removed.
- Step 2: Edge 0-3 in N_f should not be dotted.
- Step 7: Vertex 5 in *N* should have a double circle, and an edge 2-5 with flow 1 should be added to the flow graph.
- Step 7: The sets should be X = {0,1,2,3,5}, and Y = {4,6,7}.

The figure with typos corrected is included at the end of this document.

Exercise 7 (Question 7.c – 7.f can be left to the students)

Study figure 14.10 on page 444 of the text book (B&P). (Note that there are typos in at least some editions of the book: The edge (*x*1, *y*2) in the upper graph should be removed.) We now look at the duality between finding a maximum matching in the upper graph, and finding a maximum flow in the lower network (graph).

Question 7.a

Look at the following lemma, and explain why it is correct (Hint: This has also been commented on in the lectures, and it relies on the way the algorithm works):

> **Lemma** *In a network with integer capacities one can always find a flow that is both maximum and integer, and the Ford-Fulkerson-algorithm will always find such a flow.*

In other words: If the capacities are integer, we never have to split a flow so that for instance ½ goes down one edge and ½ down another to achieve a maximum flow. This means that if all capacities are 1, we get a maximum flow for the network with either full (1) or no (0) flow in each edge. Such a flow induces a subset of the edges: those with full flow.

FordFulkerson never splits an integer flow into non-integer flows, *and* proves optimality by showing a minimum cut with the same capacity as the flow.

Question 7.b

Use the lemma to explain that finding a maximum matching in the upper graph in Figure 14.10 is the same as finding a maximum flow in the lower network.

With capacity 1, flow is either 0 or 1. A flow of 1 corresponds to the edge being part of the matching.

Question 7.c

Assume that you in Figure 14.10 have the matching $\{(x2, y1), (x4, y3), (x5, y5)\}$, and show what flow f this corresponds to in the lower network.

Left to the students or the class

Question 7.d

Draw $N(f)$ (the f-derived network) for the flow from 6.c and check that looking for an f-augmenting path from s to t in this graph corresponds to looking for a (matching) augmenting path in the upper graph, with the given matching.

Left to the students or the class

Question 7.e

Use an f-augmenting path found (for instance $(x1, y1, x2, y4)$ in the graph and $(s, x1, y1, x2, y4, t)$ in the network) to augment the matching/flow, and check that these operations are duals of each other. Verify that you end up in the situation shown in the lower network in figure 14.10 (where flows are indicated).

Left to the students or the class.

Question 7.f

Draw $N(f)$ for this new flow, and show that the flow is a maximum flow by showing a cut with this capacity (4). Then use the method from Exercise 4 above to find a vertex cover of four vertices covering all edges in the upper graph, thereby showing that the matching is a maximum matching. Finally show how the cut and this vertex cover are related.

Left to the students or the class.

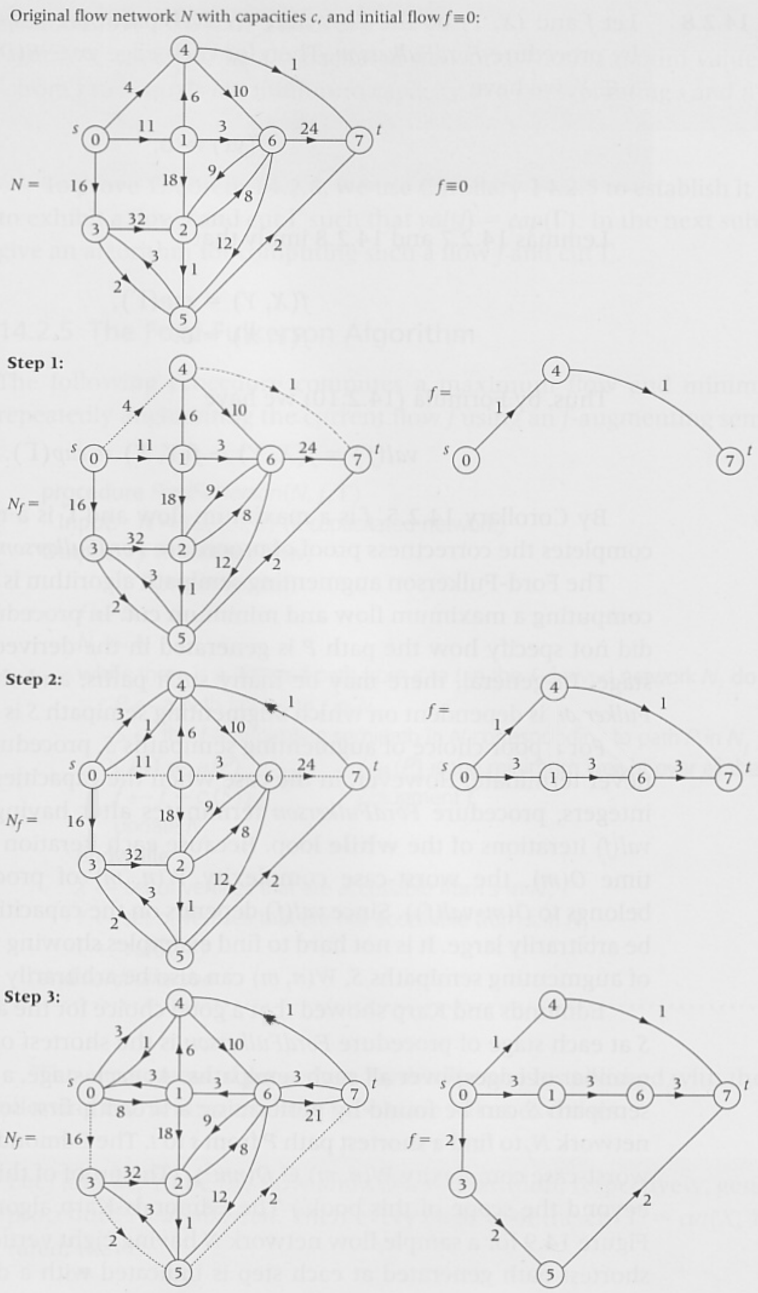
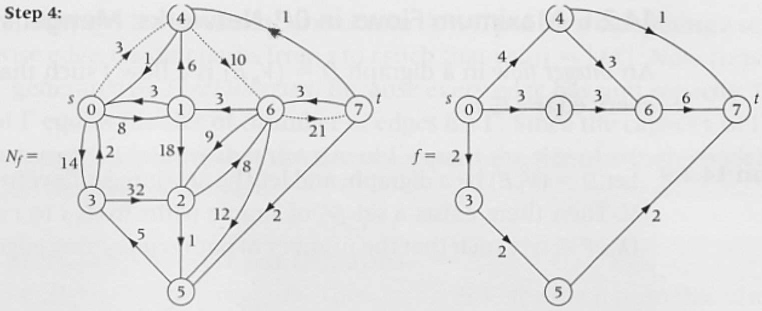


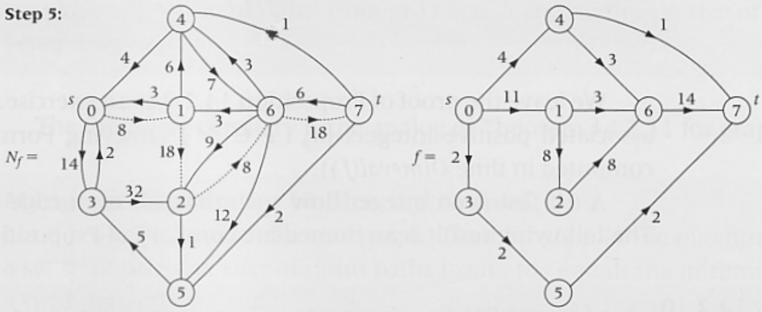
FIGURE 14.9 Action of the Edmonds-Karp algorithm for a sample capacitated network *N*.

FIGURE 14.9
Continued

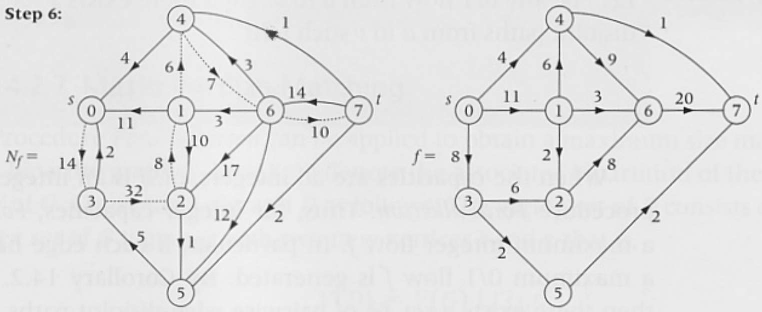
Step 4:



Step 5:

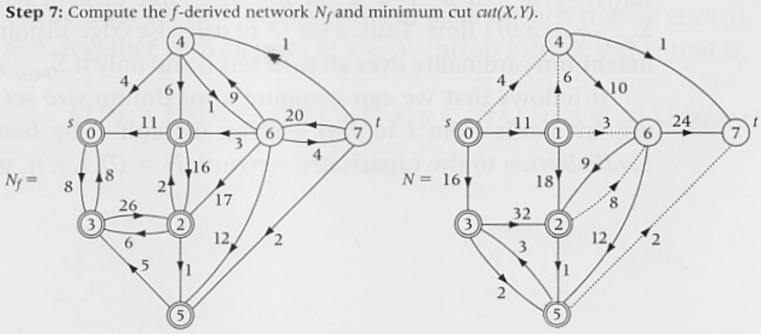


Step 6:



There are no more augmenting semipaths. The final flow f has value 23.

Step 7: Compute the f-derived network N_f and minimum cut $cut(X,Y)$.



The set $X = \{0,1,2,3,5\}$ of vertices that are accessible in N_f from the source s (marked with ◯) and the set $Y = \{4,6,7\}$ of vertices that are not accessible from s determine a cut $\Gamma = cut(X,Y)$ of capacity $c(X,Y) = 4 + 6 + 3 + 8 + 2 = 23$. Hence, we have $val(f) = 23 = cap(\Gamma)$, so that f is a maximum